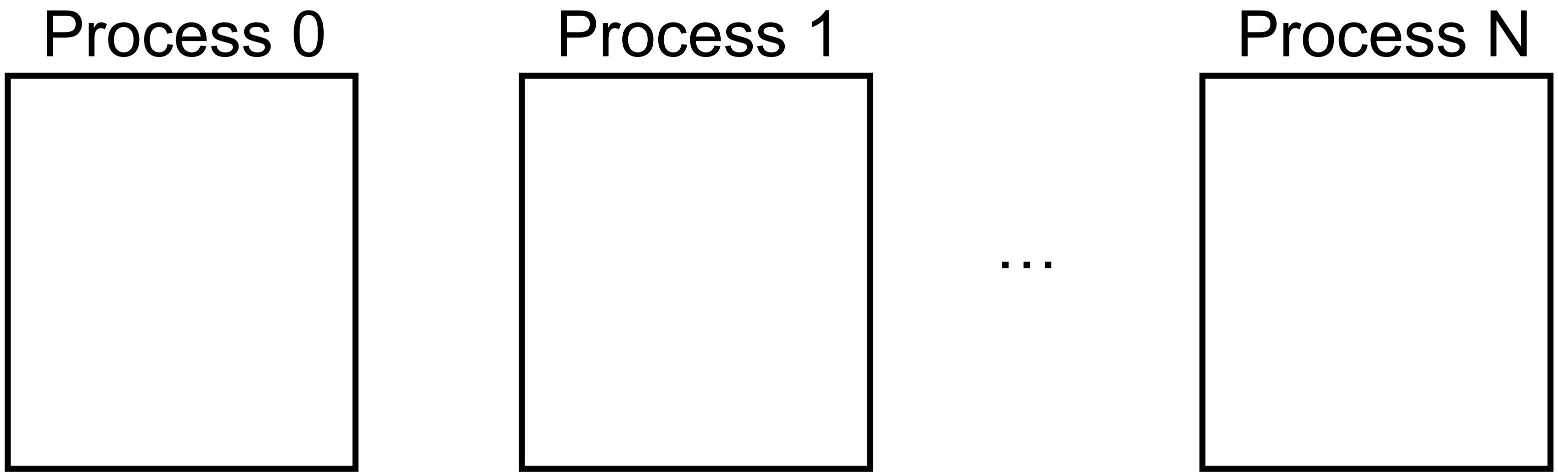
(Lack of) Overlapping of Communication and Computation in PVM and MPI

Typical development of a large parallel application

Sequential algorithm, sequential program	It's working. Let's make it faster…
Research on parallelisation methods	I'll begin with something simple…
Research on parallelisation tools	I've got a bunch of PCs. But socket programming is nothing for me…
PVM or MPI? → PVM	PVM is clumsy, no longer supported, not a standard, everyone uses MPI…
↓ MPI	
Learning MPI	How sophisticated! 200 functions!
First (very simple) parallel implementation	It's working! And it's FAST!!!
Distributed data, asynchronous comm.	I need a workaround here… Polling will do for now, I'll get rid of it later…
Optimisation	Why is it so slow?! Learning MPI tricks… Restructuring the code… What a mess! But at least it's faster (not FAST, though).

Brief introduction to (point-to-point) message passing



Each process only has access to its own memory

Each process is assigned a unique **rank** (0, 1, …, N)

Processes exchange data only via messages

There are two message passing primitives: **send** and **recv**

A message is passed between two processes (point-to-point)

A message can be sent from any process to any other one

Top ~~10~~ 5 reasons why NOT to prefer MPI over PVM

1. MPI has more than one freely available, ***quality implementation***.
2. MPI defines a 3rd party profiling mechanism.
3. MPI has ***full asynchronous communication***.
4. MPI groups are solid, efficient, and deterministic.
5. MPI ***efficiently manages message buffers***.
6. MPI synchronization protects 3rd party software.
7. MPI can efficiently program MPP and clusters.
8. MPI is totally portable.
9. MPI ***is formally specified***.
10. MPI ***is a standard***.

“1.MPI has more than 1 freely available quality implementation.”

“There are at least LAM, MPICH and CHIMP.”

All freely available implementations (and most commercial ones) are **thread-unsafe***. **This implies polling** in non-trivial (irregular) parallel applications. Polling implies an extremely high latency, non-deterministic behaviour, destroying natural structure of applications, etc.

<u>quality implementation = no polling</u>

The only currently available quality implementation of the MPI standard in this sense is **MPI/Pro** (by MPI Softtech):

- **thread-safe**
- **no polling inside**
- **not free** ($100 to $200 per process)
- still **not a quality implementation *of message-passing***

* MPICH is also thread-safe in the meantime

"3.MPI has full asynchronous communication."

"Immediate send and receive operations can fully overlap computation."

This is not true for the majority of MPI implementations (all except of MPI/Pro in 2004).

Most MPI implementations violate the **progress rule**.

Consequence:

1. P1 sends a message to P2 using Isend (immediate send)
2. P1 enters a computation which takes an unpredictably long time (say, infinitely long)
3. P2 posts a receive, matching any message from P1

The message sent in step 1 will never be delivered to P2. Therefore, there is **no overlapping of communication and computation**.

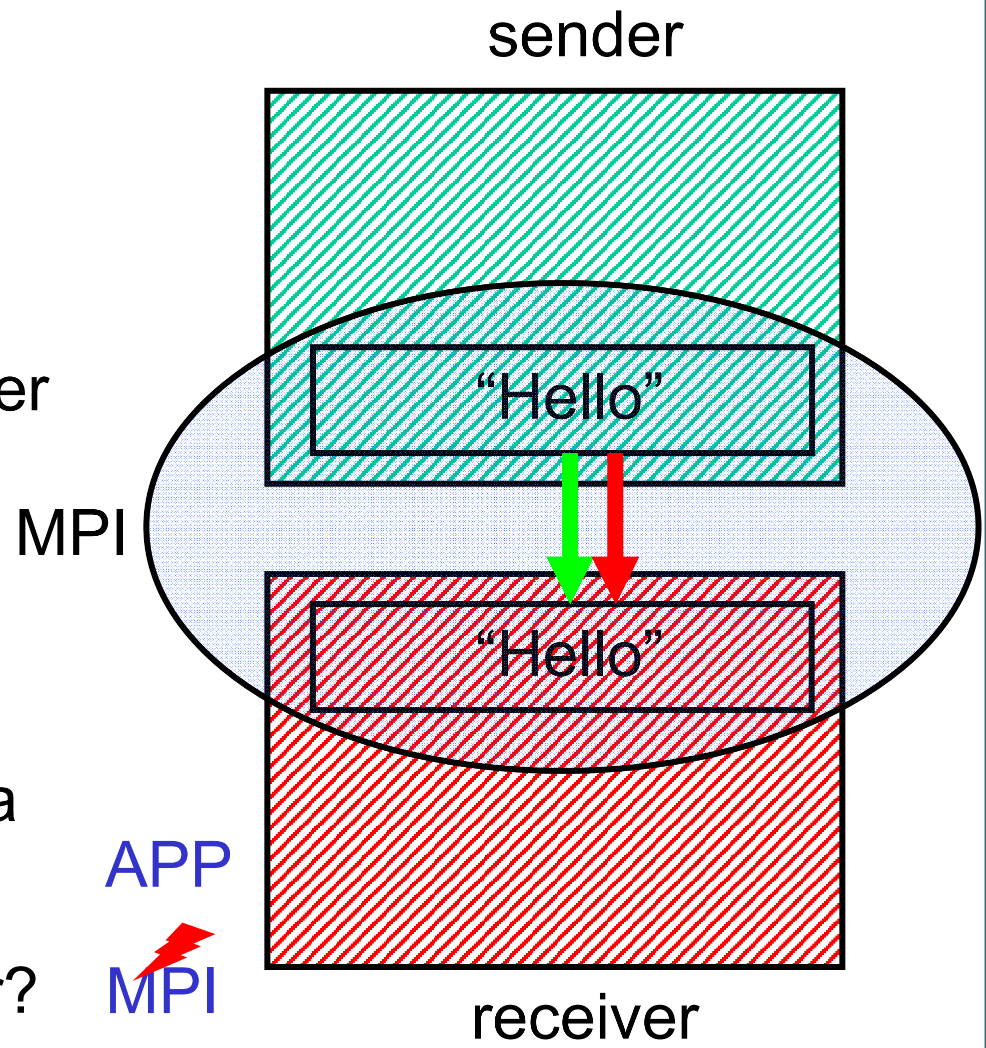
“3.MPI has full asynchronous communication.”

“Immediate send and receive operations can fully overlap computation.”

- MPI/Pro does not violate the progress rule. Nevertheless,
- **MPI/Pro does not support asynchronous communication,** because MPI/Pro implements the MPI standard
- **The MPI standard does not define asynchronous communication** as it is defined in abstract message passing models (the latter definition is more powerful)

"3.MPI has full asynchronous communication." Sending

1. Allocate a buffer
2. Pack data into the buffer
3. Send the buffer to the receiver



Questions:

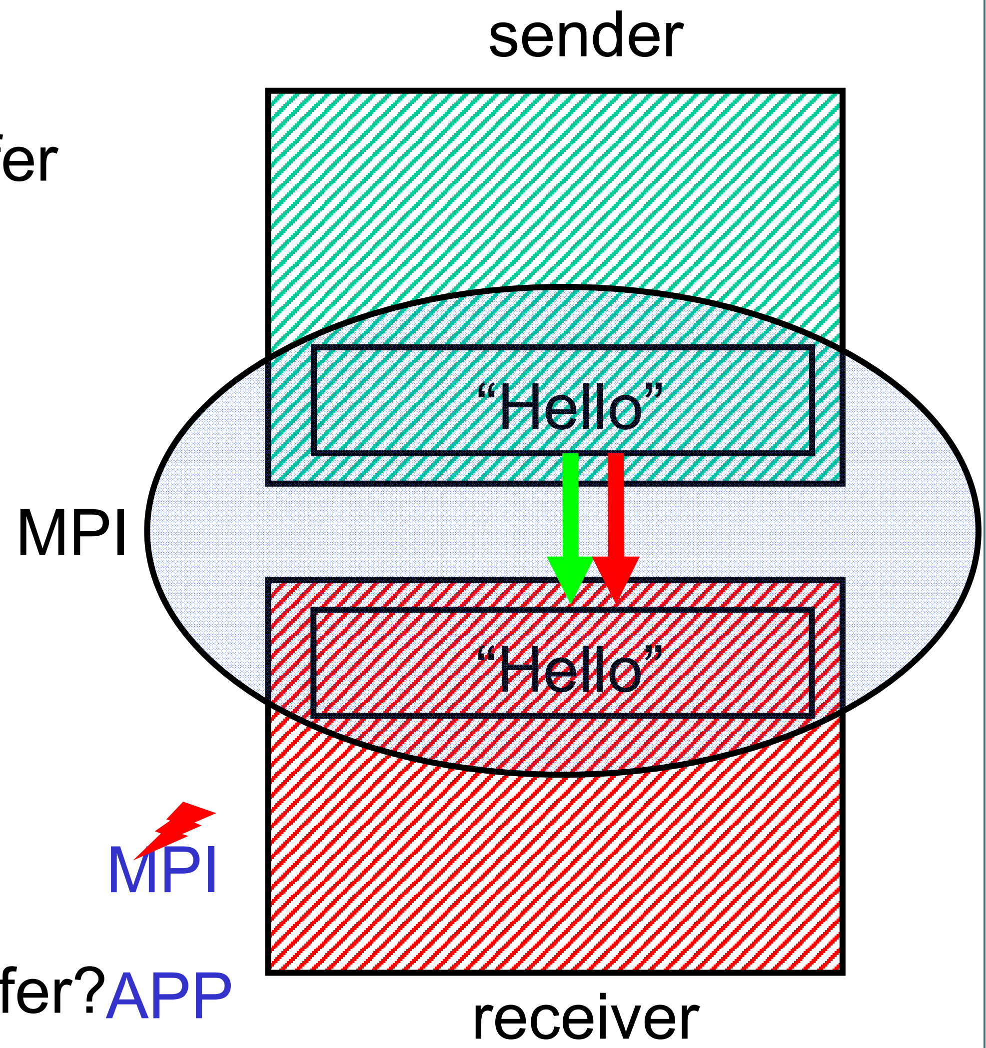
Who should decides how large a send buffer to allocate? APP

Who should free the send buffer? MPI

Blocking or nonblocking send? nonblocking is enough

“3.MPI has full asynchronous communication.” Receiving

1. Receive a message to a buffer
2. Unpack data from the buffer
3. Free the buffer



Questions:

Who should decide how large a receive buffer to allocate? MPI

Who should free the receive buffer? APP

Blocking or nonblocking recv? blocking is enough

“5.MPI efficiently manages message buffers.”

“Messages are sent and received from user data structures, not from staging buffers within the communication library. Buffering may, in some cases, be totally avoided.”

This may be true, but **this approach is wrong**.

See previous examples.

"9.MPI is formally specified."

"Implementations have to live up to a published document of precise semantics."

Since 1994 (!) there has been a quarrel among MPI developers, concerning the interpretation of the progress rule. This interpretation has never been clarified:

„If a pair of matching send and receive have been *initiated* on 2 processes, then at least one of these *operations* will *complete*, independently of other actions in the system. The *send operation* will *complete*, unless receive is satisfied by another message and *completes*. The *receive operation* will *complete*, unless the message is consumed by another matching receive that was *posted* at the same destination process."

Unclear notions: *operation, initiation, completion*. (These are **defined nowhere in the MPI standard**.)

"10.MPI is a standard."

"Its features and behaviour were arrived at by consensus in an open forum. It can change only by the same process."

MPI may be a standard in the sense that it is supported by the MPI forum. However,

The MPI standard substantially differs from (and is *provably weaker than*) abstract models for parallel and distributed computing (by Hoare, Andrews, Bernstein, Hansen and others).

The **MPI standard violates basic principles** of existing abstract message-passing models.

What cannot be done with MPI

```
p0(FILE *inp0)
{
  while(! feof(inp0))
  {
     new(m);
     m = fgetc(inp0);
     async_send(p1, m);
     printf("sent");
  }
}
```

```
p1(FILE *inp1)
{
  while(! feof(inp1))
  {
     sync_recv(p0, m);
     printf("received %c", m);
     delete(m);
     fgetc(inp1);
  }
}
```

An equivalent program cannot be written using MPI functions without breaching the bounds of the invariance thesis. The invariance thesis states: „‘Reasonable‘ machines can simulate each other with a constant factor overhead in space and a polynomial factor overhead in time.“

Kicking dogs, walking dogs on hind legs

„MPI should not need kicking (polling) to keep it moving, doing so is not good portable code... I am strongly opposed to anyone kicking their dog.“
-- Dick Treumann, IBM
(Usenet, comp.parallel.mpi, Sep 2002)

However, the MPI implementors **do** kick their dogs.
Hence, „Parallel processing is like a dog‘s walking on its hind legs. It is not done well, but you are surprised to find it done at all.“
-- Steve Fiddes (University of Bristol), with apologies to Samuel Johnson

„PVM/MPI as well as any application based on PVM/MPI should not need kicking.
I do not kick MY dogs (**I do not have to**).“
-- Tomas Plachetka
(Euro PVM/MPI, Linz, Oct 2002)

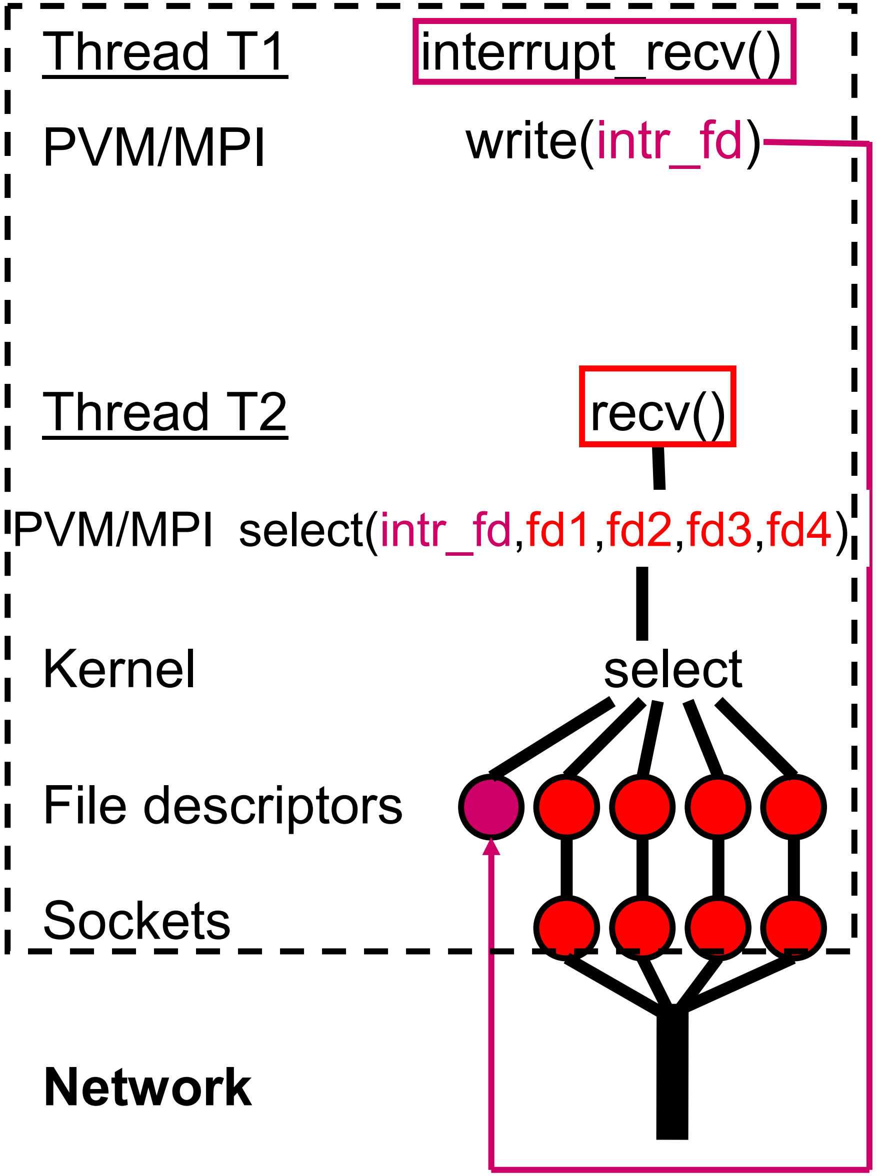
Quasi-thread-safe MPI (TP, 2002)

```
Thread T1                      Thread T2
while (not_done)               while (not_done)
  compute();                     recv();
  interrupt_recv();              handle_message();
  send();
  resume_recv();
```

PVM or MPI

```
interrupt_recv()               resume_recv()
  write(intr_fd);                write(intr_fd);

recv()
  do
    /* critical section */
    ...
    select(..., intr_fd);
    if (some fd besides intr_fd is fired)
      do_original_recv_stuff();
      interrupted = FALSE;
    else
      interrupted = TRUE;
    /* end of critical section */
    if (interrupted)
      read(intr_fd);
    read(intr_fd);
  while (interrupted);
```



What cannot be done efficiently with MPI / PVM

Non-trivial applications

Two independent activities in each process:

- T1: CPU-intensive computation which involves (occasional) communication
- T2: (fast) servicing of incoming messages which involves comunication

Examples of non-trivial parallel applications:

- distributed databases
- media servers
- application-independent load balancing libraries
- shared-memory simulation libraries
- *parallel photorealistic rendering (ray tracing, radiosity)*
- applications needing global error detection/handling, …

My thesis: „All interesting parallel applications are non-trivial.“

What cannot be done efficiently with MPI / PVM

Natural, threaded implementation

shared data

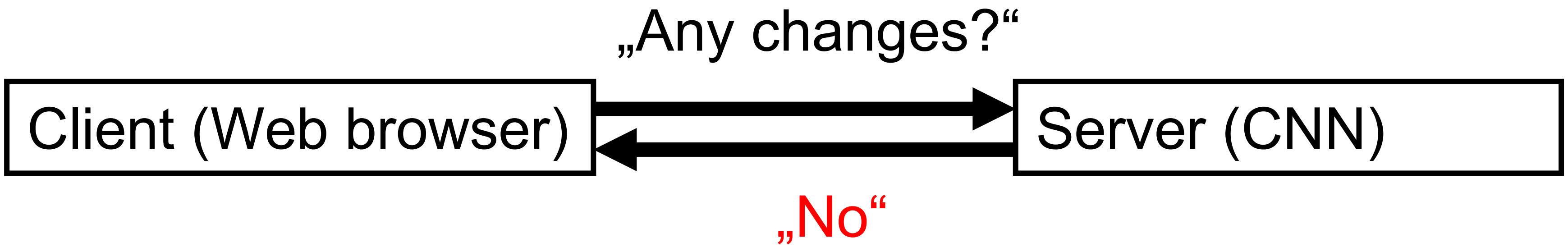
Thread T1	Thread T2
while (not_done)	while (not_done)
compute();	sync_recv();
async_send();	handle_message();

Does not work with PVM (workaround: polling)

Does not work with MPI (workaround: polling)

Note: it does not matter whether the polling is done in application or is hidden in a library

Polling: source of inefficiency



First, the **subscriber wastes an enormous amount of time and energy** asking, „Are there any changes?“ **The publisher also wastes time and effort** replying, „No, there aren´t.“ And anyone who has ever screamed at their child´s tenth repetition of „Are we almost there yet?“ in as many minutes will understand the **fundamental wrongness of this approach**.

Polling: source of inefficiency

„Any changes?“

1 hour later...

Client (Web browser) → Server (CNN)

Client (Web browser) ← Server (CNN)

SVK 4 – RUS 3

„Yes“

SVK 4 – RUS 3

Second, polling involves some **inevitable amount of latency** between the time the change occurs and the time the subscriber polls for it. On average, this latency is equal to half the polling interval. **As you lengthen the polling interval to waste fewer CPU cycles, the latency increases.**

Not only is this latency bad in and of itself, but the fact that it is **nondeterministic** (it varies from one call to another) is also a problem in designing systems.

Symmetrical THREADED-PINGPONG

working, but **inefficient** and **nondeterministic**

mutex **comm**;

Thread T1	Thread T2
while (not_done)	while (not_done)
compute();	lock(**comm**);
lock(**comm**);	arrived=probe();
send();	if (arrived)
unlock(**comm**);	recv();
	handle_message();
	unlock(**comm**);
	sleep(time);

PVM or MPI

sleep(time)≥0.02 sec

Limit of **50** calls to handle_message() per second!

Benchmark: thread-unsafe vs thread-safe

Thread-unsafe

<u>Thread T1</u>
while (not_done)
  /* compute(); */
  ~~lock(comm);~~
  send();
  ~~unlock(comm);~~

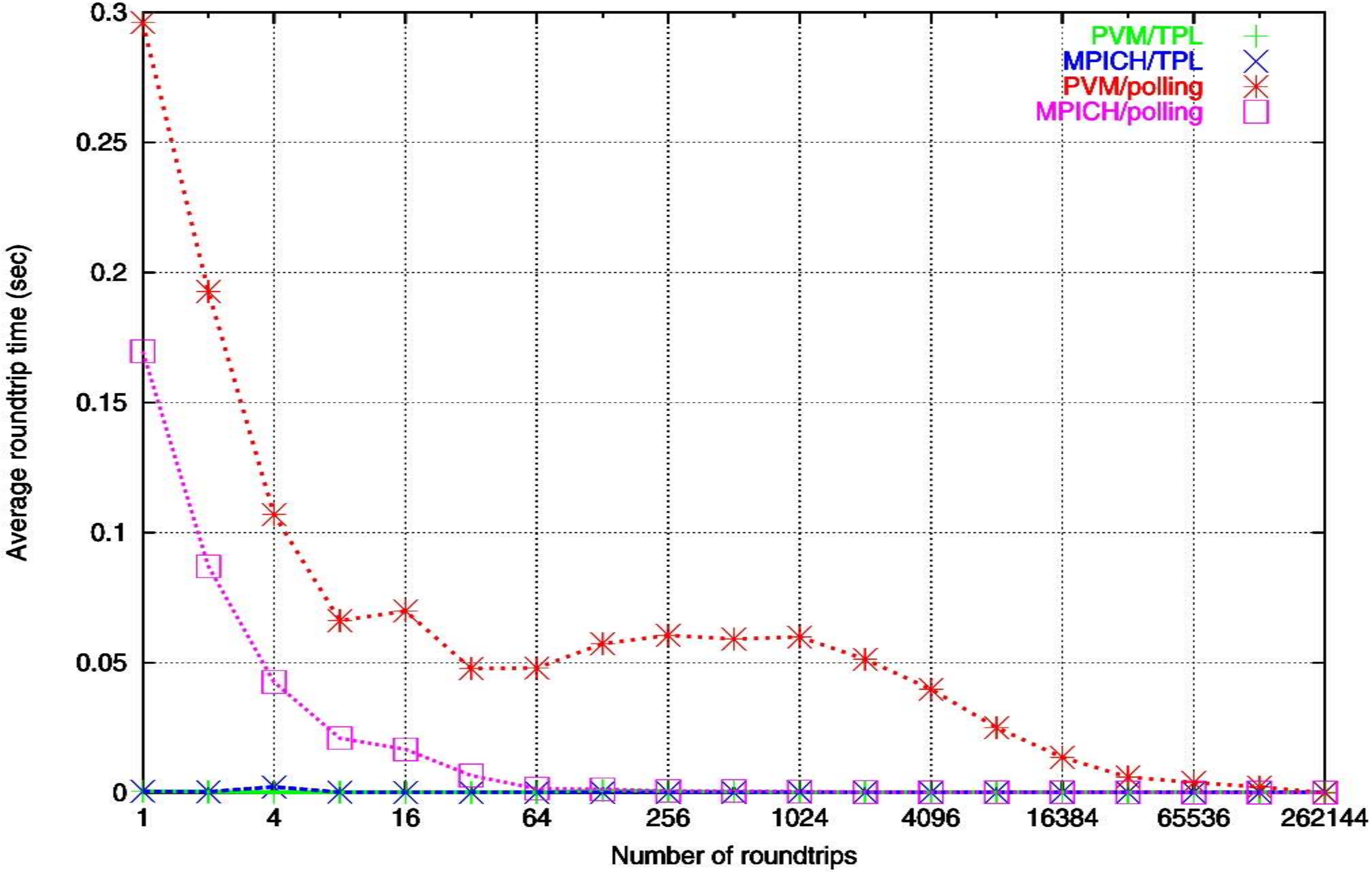
<u>Thread T2</u>
while (not_done)
  ~~lock(comm);~~
  ~~arrived=probe();~~
  ~~if (arrived)~~
    recv();
    /* handle_message(); */
  ~~unlock(comm);~~
  ~~sleep(time);~~

Thread-safe

<u>Thread T1</u>
while (not_done)
  /* compute(); */
  ~~lock(comm);~~
  send();
  ~~unlock(comm);~~

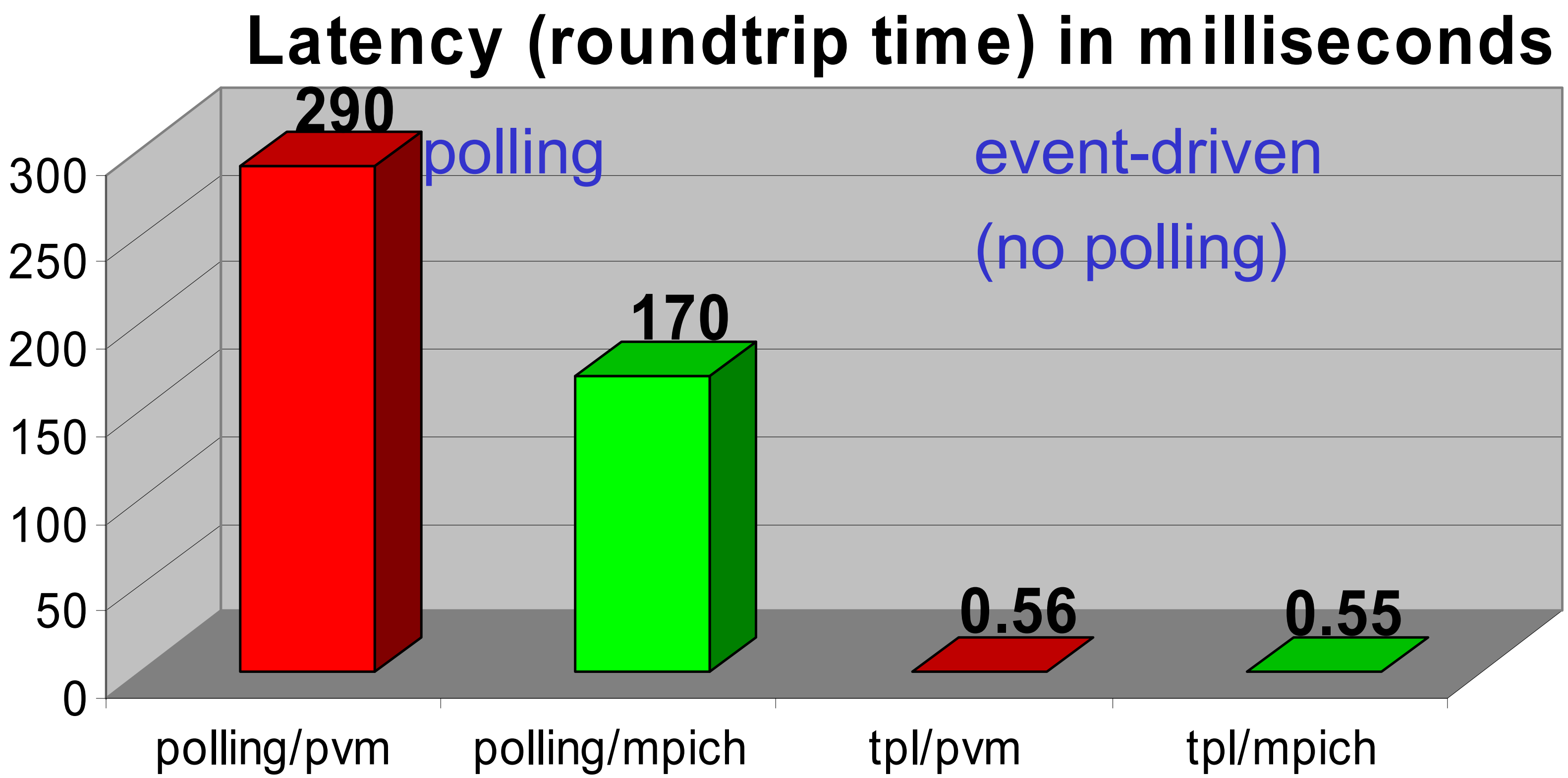
<u>Thread T2</u>
while (not_done)
  ~~lock(comm);~~
  ~~arrived=probe();~~
  ~~*while* (arrived)~~
  recv();
    /* handle_message(); */
    ~~*probe();*~~
  ~~unlock(comm);~~
  ~~sleep(time);~~

Symmetrical THREADED-PINGPONG



Symmetrical THREADED-PINGPONG

Symmetrical THREADED-PINGPONG, hpcLine, 2 nodes, TCP



Conclusions

- **Polling is evil** (non-deterministic, inefficient) **— don´t do it! In other words:** ***Don‘t use MPI, no matter whether its implementation is thread-unsafe or thread-safe***
- What to use? **A polling-free, thread-safe, portable message passing library which conforms to abstract message passing models—you have got the tools (I use TPL, my own library)**
- (Yet) not optimised polling-free TPL beats optimised polling PVM/MPI in a threaded pingpong benchmark **by factor over 300**